

Problems with Integrating Legacy Systems

Erik M. van Mulligen^{1,2,3}, Ronald Cornet¹, Teun Timmers¹

¹Department of Medical Informatics, Erasmus University Rotterdam, The Netherlands

²University Hospital Dijkzigt Rotterdam, The Netherlands

³Netherlands Institute of Health Sciences, The Netherlands

The economic and organizational impact of imposing state-of-the-art technology to the large number of proprietary legacy systems operational in most hospitals requires integrated clinical professional workstations to provide flexible encapsulation mechanisms for these systems rather than reengineering these systems to this new technology. In this paper the implications of different input/output and translation models of legacy systems for their integration into a clinical workstation is described. Examples of legacy systems that have been integrated in the HERMES clinical workstation are presented as examples of the range of difficulties one might encounter. The features that an integrated workstation should offer for integrating a broad range of legacy systems are also addressed in this paper.

INTRODUCTION

Post-factum Integration

The increasing number of publications around the clinical professional's workstation might be used to measure the maturity of hardware, software, and network technology to accomplish this dream. However, when reading these publications carefully to see what general methods are used to build such a clinical workstation, it becomes clear that most of these publications describe an actual implementation or prototype [1,2]. Although post-factum integration of legacy systems is a hard problem, it is generally seen as an essential feature for a (clinical) workstation to be of practical use [3,4,5]. The financial and organizational hazards of changing a full clinical computing environment into an open environment are too large for most medical institutions; moreover, who is going to guarantee that this new workstation technology will be up-to-date for a certain time, and that new revolutions will not occur?

Domain Specificity

Typically, the clinical workstations described in the last 3 years are very bound to the hardware,

software, and network that they have been developed for and their transfer to other domains and environments can not be realized unless one re-implements the workstation. This dependency phenomenon not only holds for clinical workstations, but is a generally recognized problem in computer science. In the last decades, a number of strategies have been developed that minimize the effect of this phenomenon and improve the independency: standardization, layering, client-server approach, and brokering (i.e. a dynamic binding between a client that requires a specific function and a selected server from a set of servers that provide that function).

Strategies to improve Independency

Although some standards may work in the construction of workstations, it is recognized that standards evolve with the progression of hard- and software. Connecting components in a workstation using standards prohibits individual components to follow new standards and thus minimizes the flexibility of the clinical computing facilities. An excellent example of how layering can be applied is the OSI network architecture. The idea is that the functionality (and the standards) of each layer can be changed without having an effect to the lower and higher layers if the layer interface remains the same. This method is often applied to the construction of network communication software. The client-server approach resembles layering, only each layer has become an independent process in the network. The client (a higher layer) requests through a specified interface (also called Application Programming Interface) a server (a lower layer) to perform a particular function. The server can be changed independently of the client and use the best known standards and technology of that time. The advantage of client-server over layering is its possible distribution through the network and the fact that a change in a server does not require the client to be recompiled (as with layering). Brokering extends the client-server approach in the sense that the client is freed from

knowing which server to address for a particular function. In addition to this, brokering can also be used to enable clients to communicate with services that have new, extended APIs by mapping the client's notion of the service API to the server's actual API.

Encapsulators

In the HERMES project we explored the above mentioned methods to see what is best suitable in constructing a clinical workstation that can be easily tailored for multiple sites and that communicates with the legacy systems present in the clinical computing environment [6,7]. In the HERMES environment, workstation components are communicating with each other through a brokering process. This broker provides all the features outlined above and uses an object database that exactly describes all requests, APIs, services, hosts, and users. One important architectural feature of HERMES is the so-called encapsulator. This piece of software interfaces with the legacy system and provides other workstation components a true open system access (with an API that is accessible through the network). In fact, this encapsulator layer (See Figure 1) makes the system dependent on a particular computing environment. All other components are only relying on XOPEN UNIX, TCP/IP, Berkeley Sockets, and X11 with OSF/Motif. In this paper, we will not extend on the full HERMES environment, but describe a number of encapsulators in detail, discuss properties and problems with the interfaces to legacy systems and try to establish a generic architecture for an encapsulator.

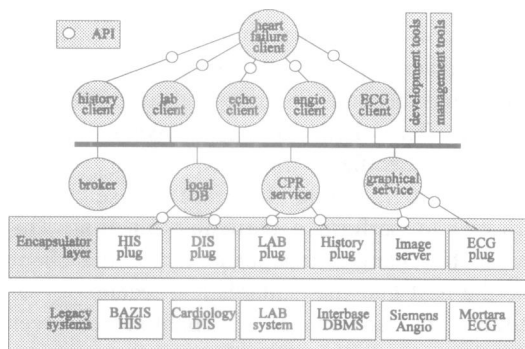


Figure 1. The HERMES integration architecture with the encapsulation and legacy system layer.

METHODS

Legacy systems can behave to the workstation encapsulator either as a process or as a session. In the case of a process, the workstation will spawn a new process of the legacy system; both the start and the end of the processing of the legacy system are controlled by the encapsulator. In the case of session orientation, the workstation has to attach to a running process often through a user authentication phase. This behavior of the legacy system has a great influence on the architecture of the encapsulator.

Interface Mechanisms

An encapsulator provides to workstation components an API that can be addressed by sending a request message to a specific network port by using the TCP/IP network protocol. Requests received by the stub are translated by the encapsulator's command generator into a series of instructions for the legacy system it is interfacing. Data that are passed with the request are reformatted by the data translator for the legacy system (see Figure 2).

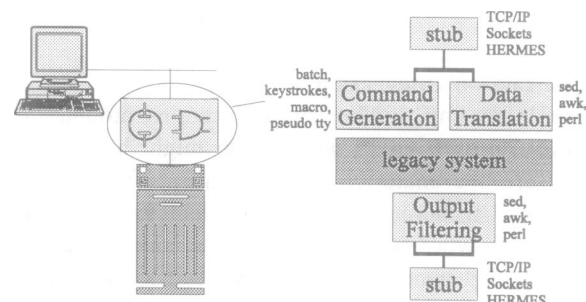


Figure 2. Architecture and modules contained in a generic legacy system encapsulator.

There are a number of options for interfacing the legacy system:

- *batch*. A batch file containing both the instructions and the data are passed to the legacy system when starting it. Results can be collected in an output file. This mode is in general only applicable for legacy systems that are started by the encapsulator as a new process.
- *keystrokes (and mouse clicks)*. The encapsulator simulates a user and generates keystrokes and possibly mouse clicks as though the user interacts with the application. The input to the legacy

- system can either go through a (Unix) pipe to the standard input of the legacy system or by setting up a pseudo-terminal that creates new I/O streams. The latter can be used for legacy systems that directly connect to (terminal) devices rather than using operating system facilities. Interfacing through keystrokes can be used for both process and session oriented legacy systems.
- *communication line*. Typically, when a legacy system is not capable of handling TCP/IP network communication, a solution is to connect the system by a RS232 communication line to a computer that is able to handle TCP/IP network communication. Typically this approach is used for session oriented legacy systems.

Output Mechanisms

Output generated by a legacy system should be filtered by the encapsulator's output filter before returning it to the requesting client. The output can either be collected in a file by (1) specifying an output filename if that is supported by the legacy system, or (2) by redirecting the screen output to a file (operating system feature) or to a filter process. Because of the absence of an end-of-output marker in the session oriented legacy systems, the output of this kind of legacy system can only be caught by a filter process that determines the end-of-output itself.

Stateless vs. Stateful

If a server process returns to an identical state after each request processed, it is called stateless communication, else stateful. Interaction with most legacy systems requires a stateful server. Therefore, it might be useful to remain connected to either a legacy system session or process for a workstation. New requests can use the results of previous requests and rely on a so-called context. However, allowing stateful encapsulators next to the usual stateless encapsulators forces additional requirements to the workstation communication architecture. A stateless encapsulator and its associated legacy system return to the initial state after each request processed. So, each client can assume the initial state when sending a request. With a stateful encapsulator however, a particular context is generated of which the client should be aware.

Exclusive threads

As a consequence, for stateful components it is essential to support a notion of exclusive communication threads. An exclusive thread

connects two workstation components and prohibits other components to use that same thread (partly) for communication. Each other component that wants to access a function in the stateful component has to set up its own exclusive thread with its own private server component (and legacy system). If it is not possible to have several legacy system processes or sessions connected at the same time, the workstation architecture should provide mechanisms for serializing requests and/or locking encapsulators (see Figure 3).

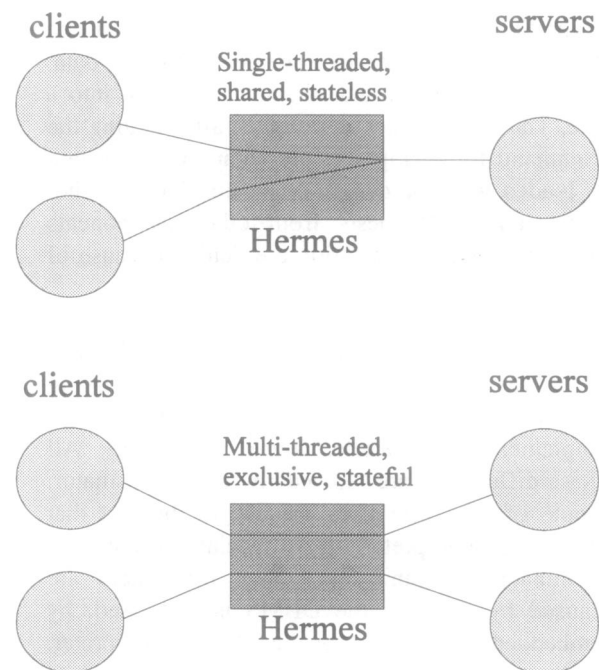


Figure 3. Examples of shared and exclusive client-server communication.

Examples of encapsulators

In the HERMES project, a number of encapsulators have been constructed for a variety of legacy systems of which the following will be outlined: a statistical package, a graphical presentation package, SQL databases, a hospital information system, departmental information system, and an ECG management system.

- Statistical package BMDP (Unix); this package is encapsulated as a batch oriented legacy system process. For each client that issues a request, a new process is spawn. The input consists of a command file and a data file. A meta-description file describes the command syntax of BMDP and is used by the encapsulator to generate a correct command file. The encapsulator maintains a list

of the variables contained in the data file together with the names as generated for BMDP (maximum of 8 characters). Output is generated by the legacy system in a file, which is filtered by the encapsulator. In this step, the encapsulator also transforms the names of the variables back to the original ones. The dependency of the output format on the values of elements in the output makes the filter rather complicated.

- Graphical spreadsheet package WingZ (Unix). Informix provides a macro language called HyperScript in which all keystrokes can be described as commands together with the data. Output is graphical and can not be written into a file. The encapsulator is statefull; after issuing the command for a graphical presentation, the system is loaded with the data and can use that for other presentations. Requests from other components for a graphical presentation are denied because of the single-user license.

- SQL databases (Unix and VMS). Although there is a reasonable standard, SQL, for querying databases, there are quite some differences in implementation (Oracle, Ingres, Interbase). All these differences are "solved" by the encapsulator. Some of the differences are due to the fact that the SQL interpreter uses special symbols to terminate commands etc. Other differences are caused by the way the DBMS is addressed: by embedded SQL or by a SQL interpreter. Although the output formats differ, they are quite regular and easy to program in an output filter. Note that it is more efficient to use a filter process than to write the selected data to a file and subsequently filter it. Most of the problems were caused by the fact that the output did not contain full attribute names due to the fact that the column width was not large enough. Special attention was paid to how to detect errors in the processing. To ensure the authentication procedures of the underlying DBMS, a special login screen has been implemented to enter these. Some DBMSs however used the (Unix) login name for authentication.

- BAZIS HIS hospital information system. The encapsulator for this system uses an RS232 connection with a micro-vax (effective data rate of 300 baud). A special communication protocol (PERI) has been developed to issue requests and retrieve a small restricted dataset. This protocol uses a number for the data-set, a number for the

patient and a byte offset for the bytes in the patient record one wants to retrieve. The encapsulator translates a local dictionary of attribute names to this data-set number and byte offset. Conditions are evaluated at the encapsulator, meaning that the complete data-set has to be retrieved. This approach is very sensitive to changes; any change in the offset of a particular variable in the patient record requires the encapsulators tables to be (manually) updated.

- Mumps departmental information system. In order to connect the Mumps system with TCP/IP, a PC was used to run DDP-DOS. With this product, the PC translates TCP/IP messages to the Mumps environment and vice versa. In the encapsulator, incoming requests are serialized and forwarded to the PC. The output is interpreted by a pipe process and returned to the client. The Mumps system provides the encapsulator with a dictionary of the requests each time it is connected. This allows for changes at the Mumps side without having to inform the encapsulator.

- Mortara ECG management system (Unix). The encapsulator for this systems runs on the same computer where the database is stored. The encapsulator uses a set of functions provided by the vendor through which the ECG database can be accessed. This approach is very efficient and allows the encapsulator to be process oriented.

Working Experiences

At this moment, a number of applications have been realized using the HERMES environment. A clinical workstation for the outpatient clinic heart failure integrates data from several clinical information systems and ECGs, angiograms and ultrasound images in a computer-based patient record. A similar approach has been followed for an application in andrology. A separate workstation application has been developed for clinical data analysis. Access to clinical information systems and commercial database management systems is combined with a range of statistical and graphical presentation functions provided by commercial statistical and presentation packages. A third type of application is developed by a third party for people involved in occupational healthcare.

CONCLUSION

The construction of encapsulators for a range of legacy systems helps to recognize what the

difficulties are in developing generic encapsulators. The following features were found to play an important role in the success of opening a legacy system with an encapsulator:

- *the operation model*; if the legacy system can be started upon request by the workstation encapsulator, the normal multi-thread model of client-server computing can be provided. If the legacy system is however an active process that needs to be attached by the encapsulator and there are only a maximum number of sessions allowed (per user), the multi-thread model will fail and a locking or serializing mechanism must be supported by the encapsulator. In the operation model, special features like communication through a serial line or a special interface computer should be accounted for.
- *the input mechanism*; the easiest systems to embed are those that use an explicit control language and support batch-file input. Generating keystrokes (and mouse-clicks) requires the legacy system to use the normal input/output channels and is strongly relying on operating system features and reliability of the keystroke generator. Typically the speed of keystroke input mechanisms is rather slow. For some legacy systems, variable labels have to be cut to a specific length and consist only of a particular set of symbols. For the encapsulator this implies that it should maintain a mapping between the actual labels and the labels as provided in the request.
- *the output mechanism*; Important to know is when the legacy system is ready with producing output. For systems that use a batch inputfile, the end of output is mostly indicated by the termination of the legacy system process. For legacy systems that generate an output file, this might be indicated by an end-of-file sign. For filtering processes that use a pipe mechanism for catching the legacy system's output, it can be difficult to determine the end of a transaction by the legacy system. Two other aspects directly complicate an encapsulator: (1) the lack of an explicit data-model that is passed with the data, and (2) dependency of the output format of the values of output variables.
- *transaction model*; if the operation of the legacy system demands an operation context, it might be efficient to leave the context intact and use it for further requests. However, this implies that the client is informed about the state of the encapsulator and excludes the HERMES feature to share an encapsulator session with different clients.

DISCUSSION

When constructing a generic encapsulator, it should be noted that these generic facilities of the encapsulator have to be able to cover all the above features. Three subparts can be clearly discerned when constructing an encapsulator: a command generator, a data translator, and an output filter. For further dissemination of this encapsulator technology, it is important to provide software developers with tools that assist them in creating the subparts of the encapsulator. And for those environments where already a host of legacy systems exist, this encapsulation technology makes it possible to have clinical workstations really integrated with these systems.

References

- [1] Hammond JE, et. al. Report on the Clinical Workstation and Clinical Data Repository Utilization at UNC Hospitals. In: Proceedings of the 18th Annual Symposium on Computer Applications in Medical Care. 1994;276-80
- [2] Young CY, Tang PC, Annevelink J. An Open Systems Architecture for Development of a Physician's Workstation. In: Proceedings of the 15th Annual Symposium on Computer Applications in Medical Care. 1991;491-5
- [3] Fournier F. The MINI Software Factory: Development of Kernel Mechanisms around Process Modeling and Software Bus Techniques. Journal of Systems Integration, 1992;2:145-67
- [4] Rossak W, Ng PA. Some Thoughts on Systems Integration: A Conceptual Framework. Journal of Systems Integration, 1991;1:97-114
- [5] Power LR. Postfacto integration technology: New discipline for an old practice. In: Proceedings of the 1st Conference on Systems Integration. 1990;4-13
- [6] Van Mulligen EM, Timmers T, Van Bommel JH. A new architecture for integration of heterogeneous software components. Methods of Information in Medicine 1993;32:292-301
- [7] Van Mulligen EM, Timmers T, Brand J, Cornet R, Van den Heuvel F, Kalshoven M, Van Bommel JH. HERMES: a health care workstation integration architecture. Int. J. Biomed. Comput. 1994;34:267-75